

# USIM Modifier V3

Tech Note

Ming



2020

# Content

Content.....	1
History .....	2
Abstract .....	3
Diagram .....	3
State machine .....	4
Logging mechanism .....	5
Extend the locale resource.....	6
Implement new plugin .....	6
Template.....	6
Language .....	7
Concept .....	7
Execute the implemented plugin .....	7
uicc Class .....	7
select .....	7
verify_pin .....	8
read_binary.....	8
update_binary.....	8
read_record.....	8
update_record.....	9
send.....	9
uicc_sel_resp Class.....	9
Convert API .....	9
convert_arguments_to_dict .....	10
convert_bcd_to_string.....	10
convert_string_to_bcd.....	10
convert_alpha_to_string.....	10
convert_dialing_number_to_string .....	11

# History

Revision	Common	Date
V1.0	Initial release	2020/5/3

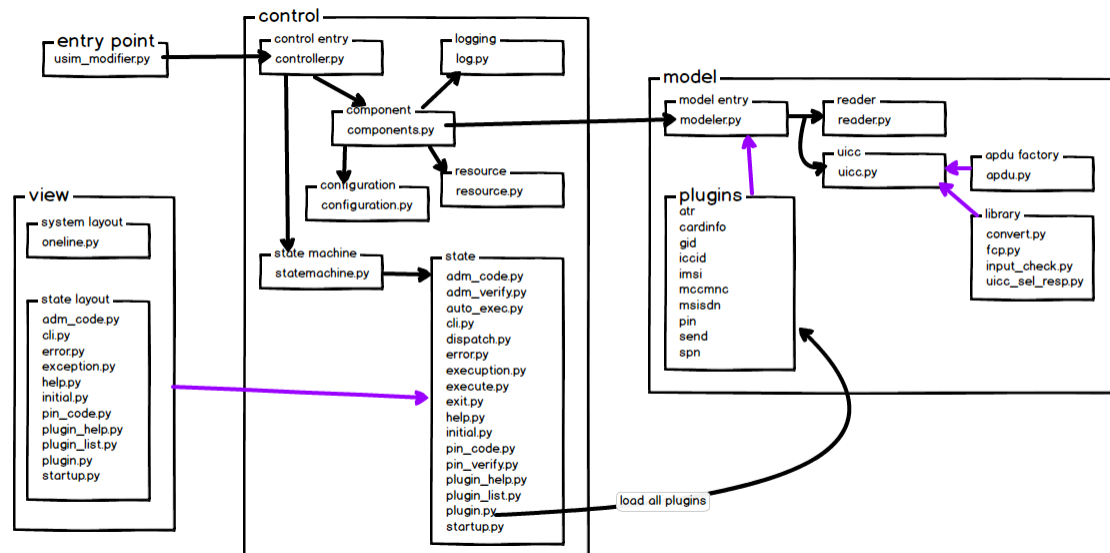
# Abstract

The document will describe the diagram, state machine, logging mechanism, extend the language resource, useful class and API for plugin develop.

# Diagram

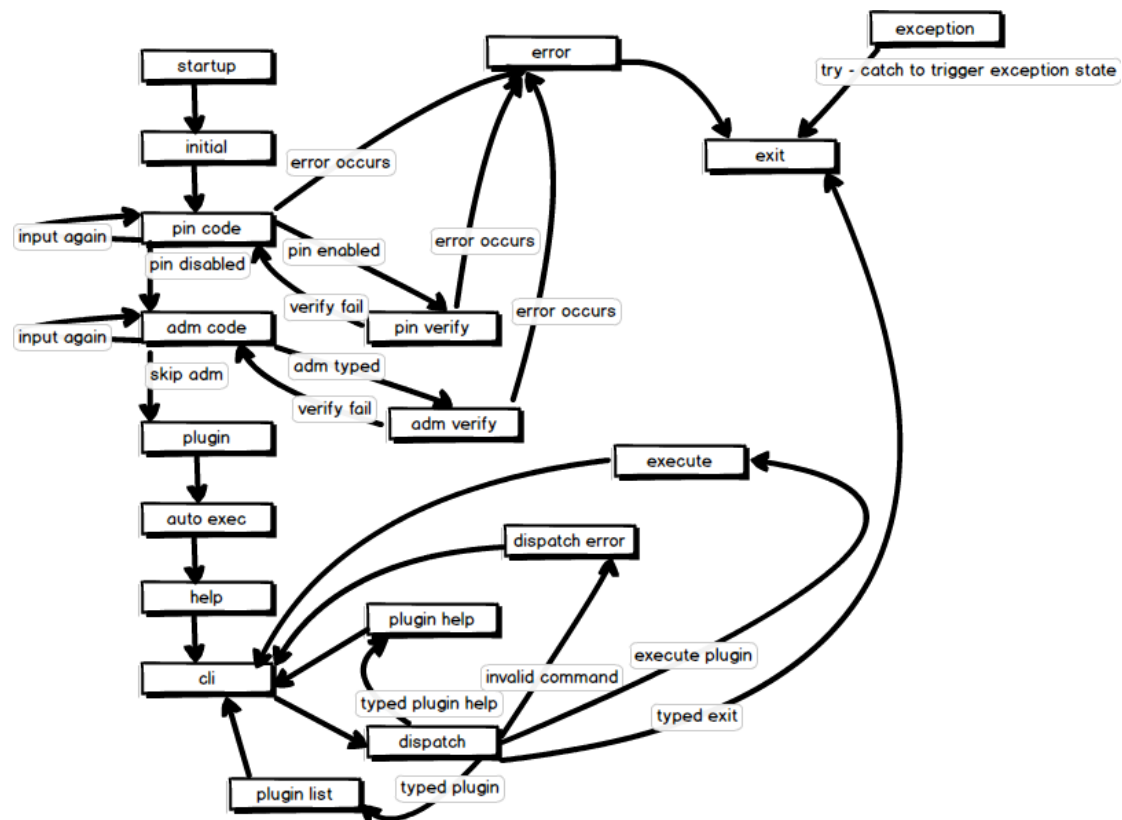
I made a strangely pattern again!

It's a very simple diagram for you get the concept of usim\_modifier\_v3.



# State machine

It's a simple state machine of "usim\_modifier\_v3", the 'statemachine.py' will handle 'self.\_\_state' variable to decide the next state, and pass 'self.\_\_arguments' as argument to next state.



# Logging mechanism

The “usim\_modifier\_v3” was supported logging mechanism, the logs will store to “logs” folder and follow “YYYYMMDD-HHMMSS.log” format as the filename, and the log format is ‘datetime’, ‘filename’, level’ & ‘content’, you can see the below sample.

05-09 11:24:34 statemachine	DEBUG	initial
05-09 11:24:34 initial	DEBUG	ENTER
05-09 11:24:34 uicc	DEBUG	TX> 00 A4 00 04 02 3F 00
05-09 11:24:34 uicc	DEBUG	RX< , 61 1F
05-09 11:24:34 uicc	DEBUG	TX> 00 C0 00 00 1F
05-09 11:24:34 uicc	INFO	ICCID: 89860009191190000108
05-09 11:24:34 uicc	DEBUG	TX> 00 A4 08 04 02 2F 00

If you want to implement the plugin by self, I suggest you can follow below **yellow line** to ‘debug’, ‘info’ or ‘critical’ logs.

```
import control.log as log

class your_class_name(base_plugin):
    def __init__(self):
        log.debug(self._class__.__name__, "debug level")
        log.info(self._class__.__name__, "info level")
        log.critical (self._class__.__name__, "critical level")
```

In ‘usim\_modifier\_v3’, we can manual edit ‘usim\_modifier.xml’ to turn off the logging mechanism.

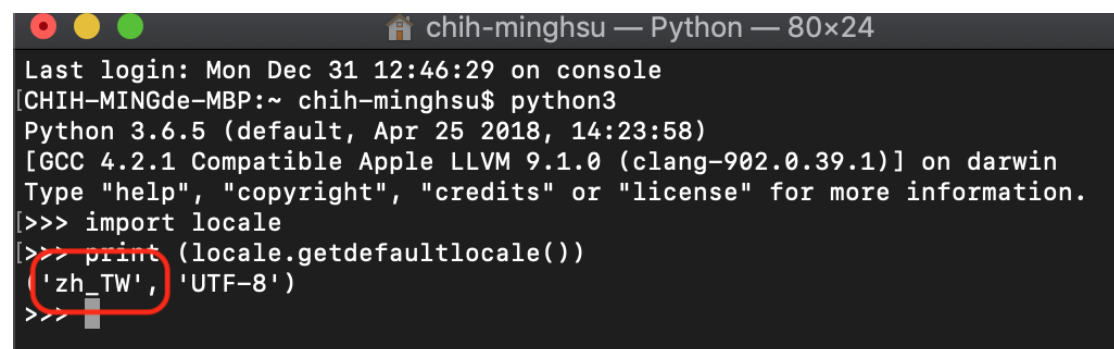
```
<?xml version='1.0' encoding='UTF-8'?>
<usim_modifier>
  <configuration>
    <log>1</log>
    <localized>1</localized>
    <pin>1</pin>
    <adm>1</adm>
    <autoexec>1</autoexec>
  </configuration>
</usim_modifier>
```

# Extend the locale resource

This project can support multiple languages (default supported 'English' & 'zh\_TW'), the locale resource is based on ".xml" file, using "English" as the default language, and you can find the "default.xml" on "control/values" folder.

If system can't figure out the resource by system locale configured, will try to search the resource of English, if still not exist, will return the error message.

To identify the system locale, please follow below steps to check the locale of your system, by example, the "zh\_TW" is locale configuration of my system.



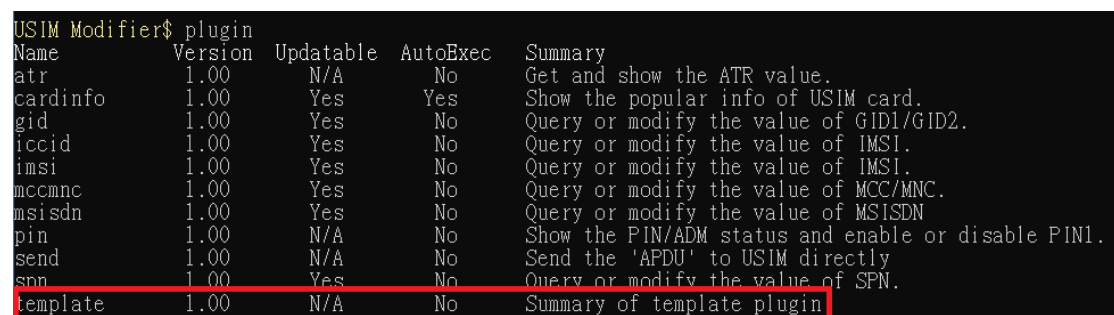
```
chih-minghsu — Python — 80x24
Last login: Mon Dec 31 12:46:29 on console
[CHIH-MINGde-MBP:~ chih-minghsu$ python3
Python 3.6.5 (default, Apr 25 2018, 14:23:58)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> import locale
[>>> print(locale.getdefaultlocale())
('zh_TW', 'UTF-8')
>>>
```

After confirmed the locale configuration, you can copy the "default.xml" and rename to "[locale configuration].xml", then translate all the English by self.

# Implement new plugin

## Template

Please copy the "docs/template" to "model/plugins" folder, and we can see the 'template' plugin already in the list.



USIM Modifier\$ plugin				
Name	Version	Updatable	AutoExec	Summary
atr	1.00	N/A	No	Get and show the ATR value.
cardinfo	1.00	Yes	Yes	Show the popular info of USIM card.
gid	1.00	Yes	No	Query or modify the value of GID1/GID2.
iccid	1.00	Yes	No	Query or modify the value of IMSI.
imsi	1.00	Yes	No	Query or modify the value of IMSI.
mccmnc	1.00	Yes	No	Query or modify the value of MCC/MNC.
msisdn	1.00	Yes	No	Query or modify the value of MSISDN.
pin	1.00	N/A	No	Show the PIN/ADM status and enable or disable PIN1.
send	1.00	N/A	No	Send the 'APDU' to USIM directly.
spn	1.00	Yes	No	Query or modify the value of SPN.
template	1.00	N/A	No	Summary of template plugin

So you need to rename the 'template' to your plugin name, include the 'template' folder & 'template.py' file, also include the class name.

## Language

Yep, the 'usim\_modifier\_v3' already support multiple languages on 'plugin' layer, in 'template' folder, we found a folder called 'values' and contain 'default.xml' file, you can base on 'extend the locale resource' chapter to implement that.

## Concept

The concept is very simple and difference with 'usim\_modifier', that's why the plugin incompatible.

In 'usim\_modifier\_v3', we just pass the arguments to plugin, it didn't handle any input & output operations, just provide frameworks to execute the plugin, and then ... forgot, it means you can request user input and display the message by your request.

## Execute the implemented plugin

Below example is show how to execute another implemented plugin, you can refer the 'cardinfo' plugin for more detail.

```
super([plugin name], self).execute_plugin([implemented plugin], arg_components)
```

## uicc Class

All communication with UISM will through "uicc" class, the plugin template already defined the 'uicc' class by 'arg\_components'.

```
uicc: uicc = arg_components.modeler.uicc
```

## select

<b>Function</b>	select(arg_file_id, arg_type=UICC_SELECT_TYPE.FROM_MF)
<b>Comment</b>	Selects a file according to the methods described
<b>Note</b>	<b>Arguments</b> - arg_file_id: file id (refer 'UICC_FILE' from 'constants.py')



	<ul style="list-style-type: none"> <li>- arg_type: select type(refer 'UICC_SELECT_TYPE' from 'constants.py')</li> </ul>
	<b>Response</b>
	<ul style="list-style-type: none"> <li>- uicc_sel_resp: response result (refer 'uicc_sel_resp.py')</li> </ul>

## verify\_pin

<b>Function</b>	verify_pin( arg_type, arg_code)
<b>Comment</b>	Verify PIN1/ADM key
<b>Note</b>	<b>Arguments</b> <ul style="list-style-type: none"> <li>- arg_type: PIN1 (0x01) or ADM (0x0A)</li> <li>- arg_code: A list of bytes as verify code</li> </ul> <b>Response</b> <ul style="list-style-type: none"> <li>- result: verify pass or fail</li> <li>- remaining: remaining count</li> </ul>

## read\_binary

<b>Function</b>	read_binary( arg_uicc_resp: uicc_sel_resp)
<b>Comment</b>	Read binary data from USIM file
<b>Note</b>	<b>Arguments</b> <ul style="list-style-type: none"> <li>- arg_uicc_resp: depend the select() response to read binary data</li> </ul> <b>Response</b> <ul style="list-style-type: none"> <li>- resp: return a list of bytes (The 'None' mean the read fail)</li> </ul>

## update\_binary

<b>Function</b>	update_binary(arg_content)
<b>Comment</b>	Update binary data to USIM file
<b>Note</b>	<b>Arguments</b> <ul style="list-style-type: none"> <li>- arg_content: Bytes of data to be update</li> </ul> <b>Response</b> <ul style="list-style-type: none"> <li>- The 'ERROR.NONE' mean the update success, else are fail.</li> </ul>

## read\_record

<b>Function</b>	read_record(arg_idx, arg_uicc_resp: uicc_sel_resp)
<b>Comment</b>	Read data from assigned record number
<b>Note</b>	<b>Arguments</b>

	<ul style="list-style-type: none"> <li>- arg_idx: Record number</li> <li>- arg_uicc_resp: depend the select() response to read record data</li> </ul> <p><b>Response</b></p> <ul style="list-style-type: none"> <li>- resp: return a list of bytes (The 'None' mean the read fail)</li> </ul>
--	---

## update\_record

<b>Function</b>	update_record(arg_idx, arg_content)
<b>Comment</b>	Update data to assigned record number
<b>Note</b>	<p><b>Arguments</b></p> <ul style="list-style-type: none"> <li>- arg_idx: Record number</li> <li>- arg_content: Bytes of the data to be update</li> </ul> <p><b>Response</b></p> <ul style="list-style-type: none"> <li>- The 'ERROR.NONE' mean the update success, else are fail.</li> </ul>

## send

<b>Function</b>	send(arg_apdu_cmd)
<b>Comment</b>	Send apdu to USIM directly
<b>Note</b>	<p><b>Arguments</b></p> <ul style="list-style-type: none"> <li>- arg_apdu_cmd: A list of bytes</li> </ul> <p><b>Response</b></p> <ul style="list-style-type: none"> <li>- response: Response APDU from USIM</li> <li>- sw1: 1st byte of status word.</li> <li>- sw2: 2nd byte of status word.</li> </ul>

## uicc\_sel\_resp Class

In 'usim\_modifier\_v3', after select USIM filed by uicc.select() function, it will return the 'uicc\_sel\_resp' class, except the 'fcp response', 'sw1' & 'sw2' data, you can get the 'pin status', 'ef type', 'record count' & 'data length' by file type.

## Convert API

The "pyscard" package already supported some convert method but still not enough for us, we provide some convert function to meet more requirement.

## convert\_arguments\_to\_dict

<b>Function</b>	convert_arguments_to_dict(arguments='')
<b>Comment</b>	Convert the argument to dict type
<b>Usage</b>	from model.plugin.api.convert import convert_arguments_to_dict
<b>Example</b>	<pre>argument_string = ('name="super star" num="+12345678"') dict = convert_arguments_to_dicts(argument_string) print (dict) → { 'name': "super star",     'num': "+12345678"}</pre>

## convert\_bcd\_to\_string

<b>Function</b>	convert_bcd_to_string(bytes=[])
<b>Comment</b>	Convert the BCD bytes array to string
<b>Usage</b>	from model.plugin.api.convert import convert_bcd_to_string
<b>Example</b>	<pre>bcd_vals = [0x98, 0x68, 0x00, 0x90, 0x91, 0x11, 0x09, 0x00, 0x10, 0x80] bcd_string = convert_bcd_to_string(vals) print (bcd_string) → '89860009191190000108'</pre>

## convert\_string\_to\_bcd

<b>Function</b>	convert_string_to_bcd(string='')
<b>Comment</b>	Convert the string to BCD array
<b>Usage</b>	from model.plugin.api.convert import convert_string_to_bcd
<b>Example</b>	<pre>bcd_string = "89860009191190000108" bcd_vals = convert_string_to_bcd(bcd_string) print (bcd_vals) → [0x98, 0x68, 0x00, 0x90, 0x91, 0x11, 0x09, 0x00, 0x10, 0x80]</pre>

## convert\_alpha\_to\_string

<b>Function</b>	convert_alpha_to_string(bytes=[])
<b>Comment</b>	Convert the bytes array of Alpha Identifier to string (Ex: EF_ADN)
<b>Usage</b>	from model.plugin.api.convert import convert_alpha_to_string
<b>Example</b>	<pre>alpha_vals = [0x4D, 0x41, 0x49, 0x20, 0x54, 0x45, 0x53, 0x54, 0xFF, 0xFF]</pre>

	<pre>alpha_string = convert_alpha_to_string(alpha_vals) print (alpha_string) → 'MAI TEST'</pre> <p><b>PS. The function only support ASCII code.</b></p>
--	---

## convert\_dialing\_number\_to\_string

<b>Function</b>	convert_dialing_number_to_string(bytes=[])
<b>Comment</b>	Convert the bytes array of dialing number to string (Include TON & NPI)
<b>Usage</b>	from model.plugin.api.convert import convert_dialing_number_to_string
<b>Example</b>	<pre>num_vals = [0x81, 0x90, 0x82, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF] num_string = convert_dialing_number_to_string(num_vals) print (num_string) → '0928000000'</pre>